<알고리즘 요약 정리본>
5장_3 B-트리(B-Tree) 삽입과 삭제

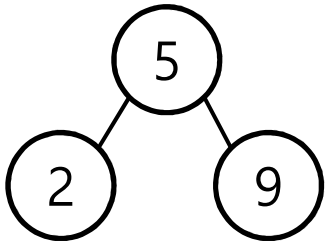
이제 알고리즘의 탐색 트리 부분에 대해서 마지막으로 들어오고 있다. 이번에는 데이터베이스 인덱스의 꽃인 B-트리에 대해서 어떤 방식으로 돌아가는지에 대해서 공부를 해보고, 이의 삽입 알고리즘과 삭제 알고리즘에 대한 작동에 대해서 알아보도록 하겠다.

0. 디스크에서 쓰이는 B-트리

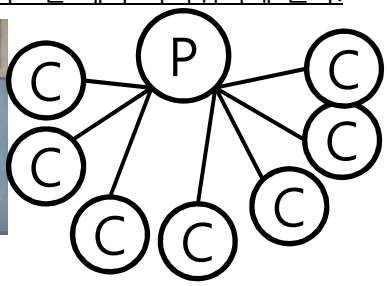
우선 HDD(Hard Disk Drive)은 하드 디스크라는 사실에 대해서 다들 알 것이다. 그렇지만 RAM(Random Access Memory)라는 녀석보다 느릴 수 밖에 없다. 한 10만 배 정도로 느리다든데 그 이유는 HDD에서 한 바이트씩 읽는 것이 아니라 블록 단위로 읽고 쓰는 점이 있기 때문이다.
이처럼 하드디스크에서 저장하는 트리 자료구조에서 노드의 크기는 블록의 트리와 일치를 하도록 설계를 하는 점에 대해서 느낄 수 있을 것이다. 이러한 원리가 바로 B-트리로 칭할 수 있겠다.

1. 외부 검색 트리와 다진 트리

우리가 외부에서 검색을 하는 경우에 효율적으로 검색을 할 수 있도록 이용하는 원리에 대해서는 외부 검색 트리를 이용하게 되는데 이 녀석은 분명 하드 디스크에 이미 저장이 되어 있는 상태에서 이용을 하게 될 것이다.
여기서 다진 트리에 대해서 알아둘 필요가 있다. 우리가 여태껏 이진 트리에 대해서 살펴보게 되었는데 이진 트리는 아시다시피 자식들이 2명 이내로 형성이 되는 트리로 알 수 있었다. 마치 짱구라고 친다면 아래처럼 짱구와 짱아로 볼 수 있겠다.



이제 B-트리에서 주로 쓰이게 되는 다진 트리에 대해서 알아보도록 하자. 다진 트리는 더더욱 쉽게 이야기 하자면 자식 노드의 수가 2개보다 더 크면 이를 다진 트리라고 볼 수 있다. 마치 짱구에서 짱구가 양산형으로 낳아진 경우에 대해서 생각을 할 수 있겠다. 그렇지만 다진 트리를 이용하게 된다면 시간 복잡도가 log n이 나오는 것이 아니라 자식의 수에 따라서 시간 복잡도가 달라질 수가 있는 대신에 알고리즘의 난이도는 꽤나 어려워지게 된다.

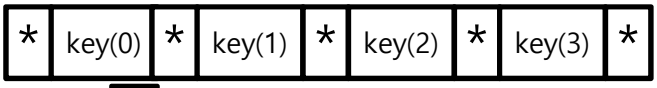


3. B-트리

B-트리는 외부 다진 검색 트리이다. 이는 하드 디스크에 저장된 상태에서 이용이 되는데, 이 녀석은 또한 필자가 환장하는 데이터베이스의 인덱스 구현에서도 B-트리를 적용할 수가 있다. 여기서 B-트리의 노드 크기는 하드디스크의 블록 크기로 볼 수 있는데 B-트리의 노드의 구조는 어떻게 되는지에 대해서 위에서 살펴보도록 하겠다.

-> 오른쪽 위로 ->

3-1) B-트리 노드의 구조로는?



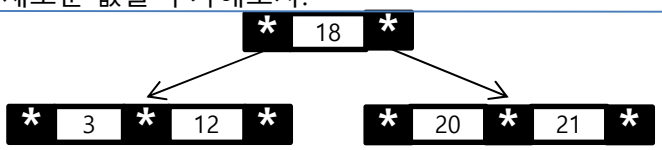
여기서  은 각 키에 이전 값들이 저장된 서브트리를 가리키거나 다음에 있는 것들은 각 키의 다음 값들이 저장된 서브트리를 가리키는 포인터로 살펴볼 수 있겠다. 일단 여기에는 노드에 들어있는 키의 수가 4개이니깐 자식 노드의 수는 5개로 살펴볼 수 있겠다. 그니깐 키의 수에 따라서 자식 노드의 수는 1씩 늘어나게 된다고 보면 되겠다.
그리고 key(0), key(1)...은 아시다시피 레코드의 키 값이다. 일단 우리는 여기에 숫자만 대입을 하고 넘어가는 방식으로 볼 것이다. 원래 데이터베이스에서 키는 대부분 기본 키(Primary Key)를 두고 본다. 그래서 기본 키를 통해서 어느 위치에 있든 간에 같은 높이로 유지를 함으로서 데이터를 신속히 찾아 나가는 원리를 인덱스라고 하는데 인덱스가 사실상 많아지면 탐색의 방향이 꼬아지는 악영향을 주니 적당히 쓰는 것이 약이겠다.

여기서 중요한 것은 B-트리에 대한 특성을 조금 살펴봐야겠다.
-> B-트리는 균형이 잡힌 다진 트리로 유지를 하기 위해서 자식의 수가 k인 경우에 k/2 ~ k까지로 제한을 둬야 된다.
용량의 낭비로 인한 언더 플로우가 발생을 하기 때문에 이에 대해서 방지를 해서 다른 자식들과 병합을 함으로서 진행이 되어야겠다.
-> 모든 리프 노드는 같은 깊이를 가진다.
어느 자식만 깊고, 어느 자식만 얕고 그런 거 없다. 오버플로우와 언더플로우를 통해서 이를 방지를 하고, 같은 높이에서 데이터를 찾는데 있어서 공평성을 제공해야 하는 노드가 바로 B-트리로 볼 수 있겠다..

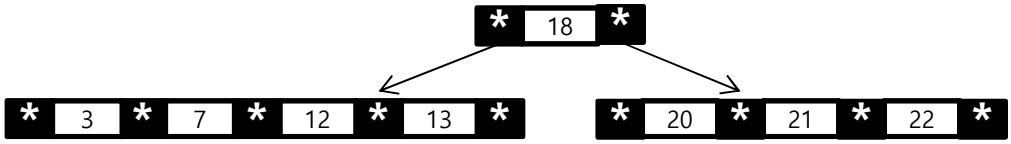
3-2) B-Tree 삽입 작동 원리
참고로 B-Tree는 자바로 구현을 하기에는 에바이다. 그래서 우리는 이러한 알고리즘이 어떻게 돌아가는지에 대해서만 알고 넘어가는 것으로 공부할 수 밖에 없다 애석하게도... 그래서 이번에는 요약 정리를 넓힌 시점을 통해서 이 알고리즘이 어떻게 돌아가는지에 대해서 공부를 해둘 필요가 있다.
<조건 1> 노드에 저장 가능한 키의 최대 수는 4로 제한.
<조건 2> 언더 플로우에 대해서는 삭제 알고리즘에서 고려해보기로 하고 삽입 알고리즘이 어떻게 돌아가는지에 대해서만 공부를 해보는 것으로 종결하겠다.

<추가 정보>
B트리 알고리즘을 뷰어로 보고 싶은 사람은 아래에 있는 홈페이지를 통해서 살펴보면 도움이 되겠다.
https://www.cs.usfca.edu/~galles/visualization/BTree.html
그렇지만 단점은 강의 노트에서 살펴본 최대 노드의 수에 대해서 바꿔서 봐야(만일 강의노트에서 최대 노드의 수가 4이면 여기서는 Max. Degree = 5를 선택해야 된다...) 된다는 점이 단점이다. 그리고 언더 플로우에 대해서는 구현이 안 되어있다는 점에 대해서도 아쉽게 생각을 한다. 그리고 수업시간에 했던 알고리즘과 차이가 있기 때문에 이에 대해서 공부를 하면서 주의를 요하길 바라면서 또한 우리가 설명하는 방법으로서는 교수님이 이야기했던 알고리즘으로 작동 방법에 대해서 연습 문제를 풀어보는 방식으로 알아보겠다.

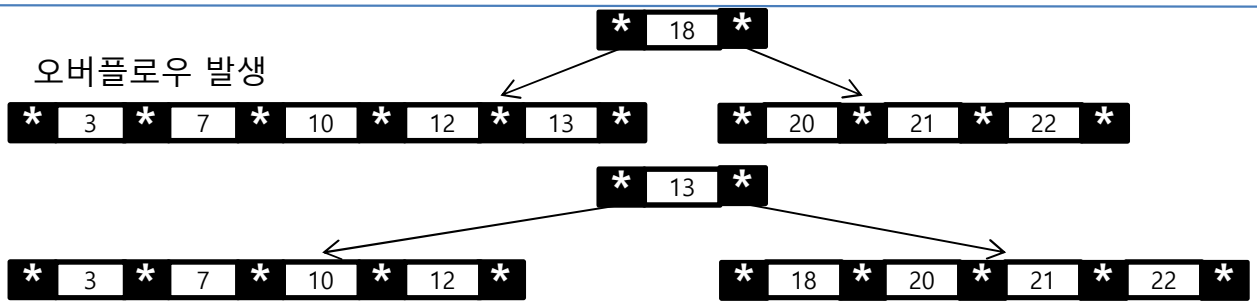
1단계) 20, 3, 18, 12를 삽입하게 된다면...
일단 3부터 20까지 오름차순으로 만들어지게 된다. 일단 삽입 할 때에 있어서 최소 노드의 수에 대해서는 굳이 생각하지 안 해도 장땡이다. 이제 새로운 값을 추가해보자.



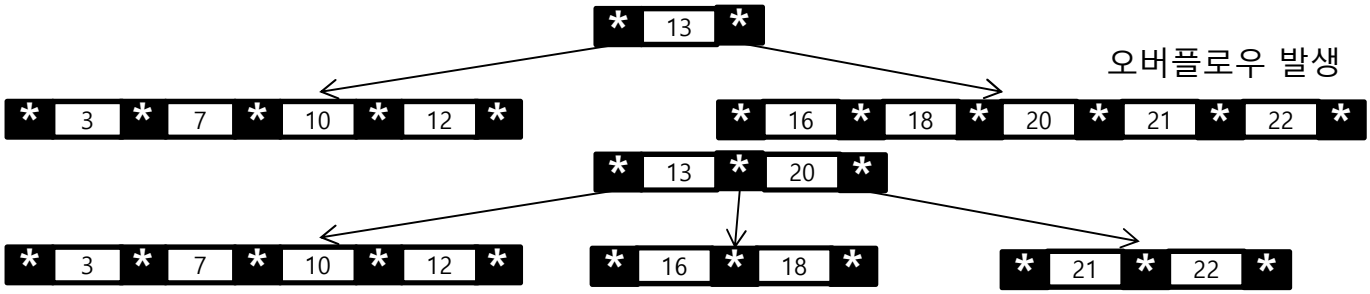
2단계) 21를 삽입시켜 본다면
21를 넣게 된다면 오버플로우가 발생하기 때문에 이에 대해서 분할을 거치게 된다. 여기서 분할하는 방법에 대해서는 중앙 인덱스를 통해서 2개의 블록으로 분할을 하는 과정으로 보면 되겠다. 그럼 18의 좌측은 3, 12가 남고, 18의 우측엔 20과 21이 남게 된다.



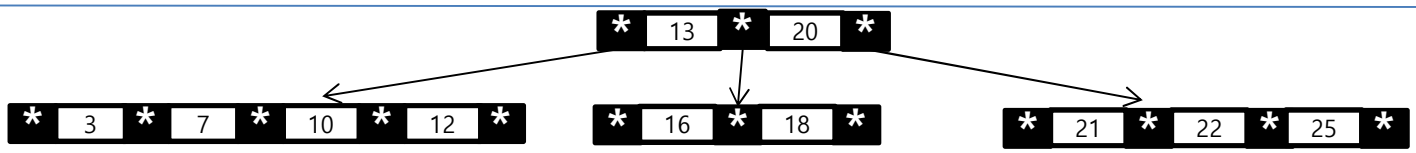
3단계) 13, 22를 삽입시켜보면
13과 22는 공간이 여유 있어서 바로 옆에 넣으면 되겠다. 18보다 작은 값은 왼쪽에, 큰 값은 오른쪽에 보내는 원칙은 그대로이니 이에 대해서 알아두고 넘어가길 바란다.



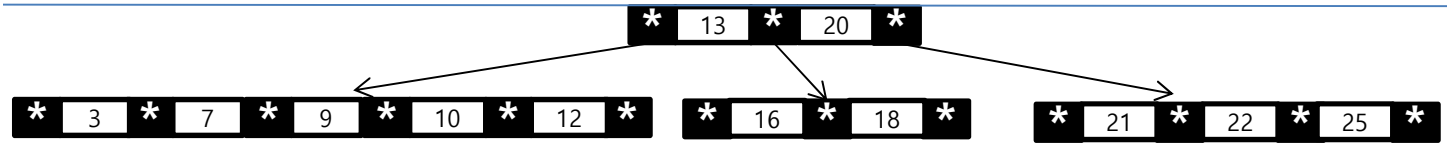
4단계) 10을 삽입 시켜본다면...
분명 좌측 서브트리에서 오버플로우가 발생을 하게 된다. 그래서 13이란 녀석을 일단 루트로 보내고 봐야겠다. 그러면 우측 서브트리에 공간이 하나 남기 때문에 18이란 녀석은 우측 서브트리로 보내지게 되면서 13이 루트 노드에 남게 된다.



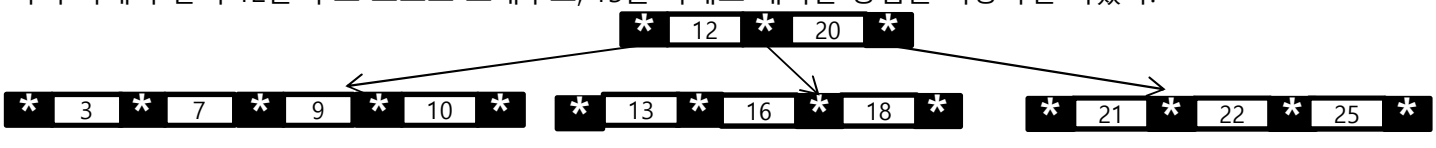
5단계) 16을 삽입하게 된다면...
우측 서브트리에 대해서 오버플로우가 나오기 때문에 분할의 필요가 있겠다. 여기서 20이란 녀석이 루트 노드로 올라오게 되면서 이에 대해 만족을 시키고 넘어간다. 공간이 전부 부족한 경우에는 분할을 하는 방법으로서 중앙 인덱스를 이용해서 분할을 할 수 있게 된다.



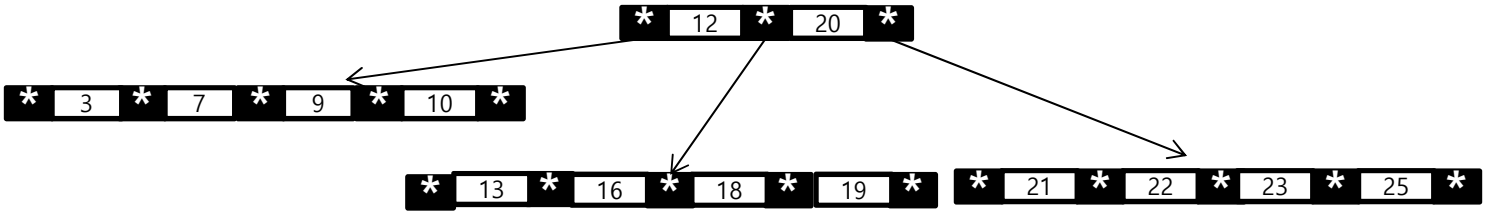
6단계) 25를 삽입한다면... 그냥 삽입하고 끝내면 된다.



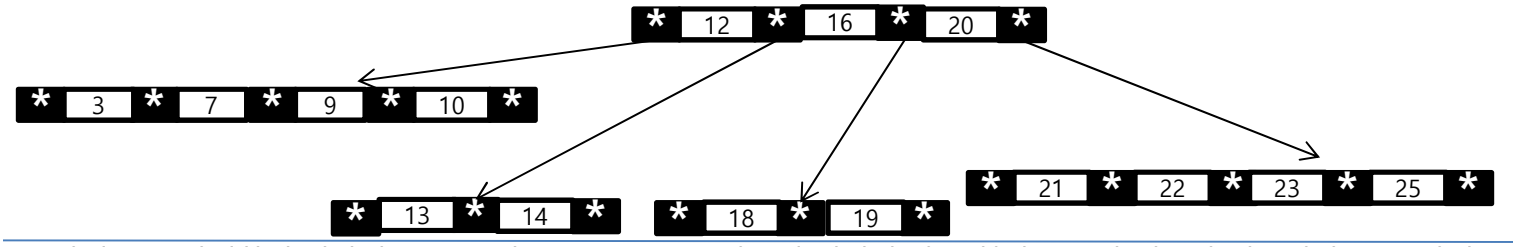
7단계) 9를 삽입하게 된다면 첫 번째 서브트리에서 그대로 오버플로우가 된다. 하지만 형제 노드께서 여유가 있다고 하니 아래와 같이 12를 루트 노드로 보내주고, 13을 아래로 내리는 방법을 이용하면 되겠다.



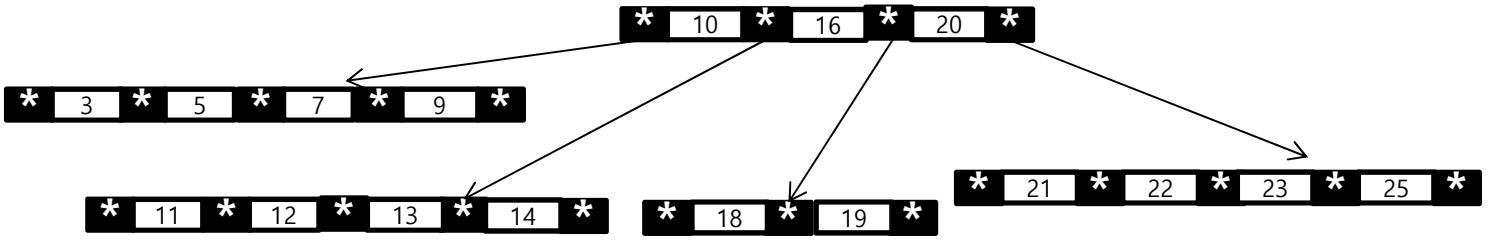
8단계) 23, 19을 삽입하게 된다... 음... 어... 그냥 삽입하면 끝이다.



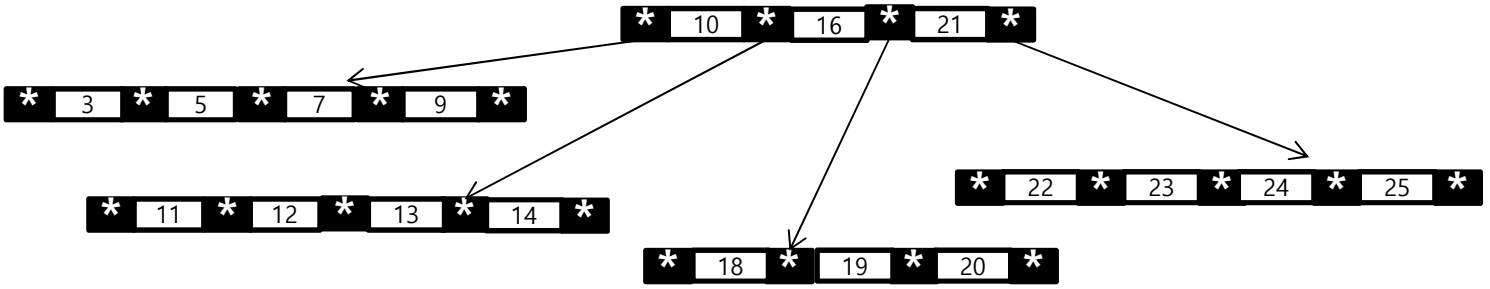
9단계) 14를 삽입하게 된다면 일단 형제 노드의 여유가 없으니 16이란 노드가 루트 노드로 올라가게 된다..



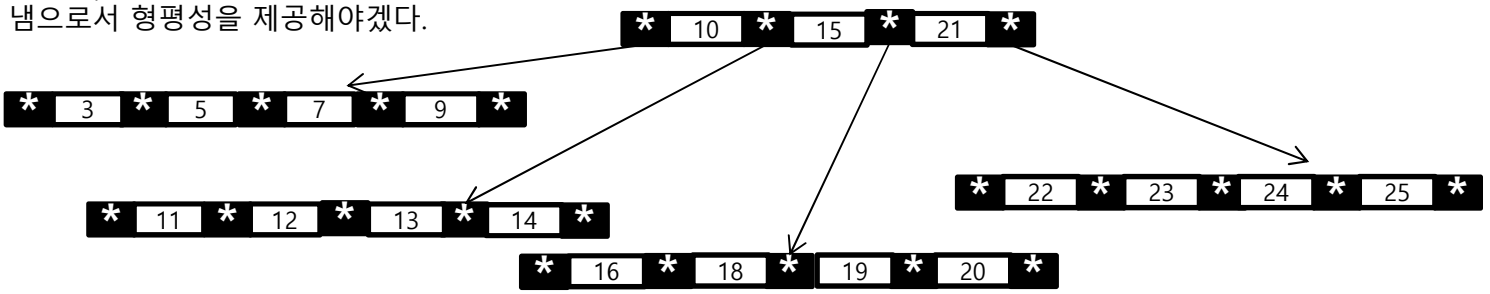
10단계) 5를 삽입하게 된다면 10 노드가 루트 노드로 올라오게 되면서 다음 형제 노드가 여유가 있으니깐 12를 삽입을 하고 끝나게 된다. 그리고 11도 그냥 삽입을 함으로서 종결을 내면 된다.



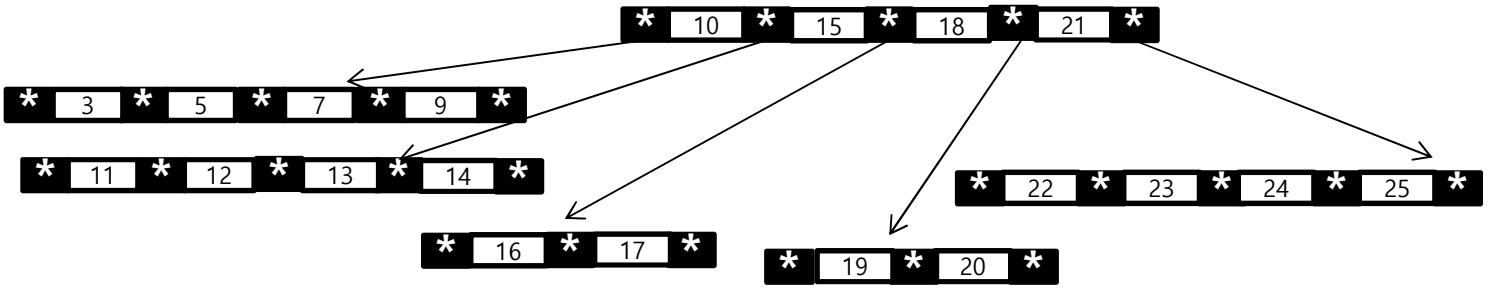
11단계) 24를 삽입하게 된다면 맨 오른쪽 끝의 형제 노드(18부터...)가 여유가 있기 때문에 20을 19 옆으로 보내고 21를 루트 노드로 올려야 된다. 그럼 아래와 같이 완성이 되겠다.



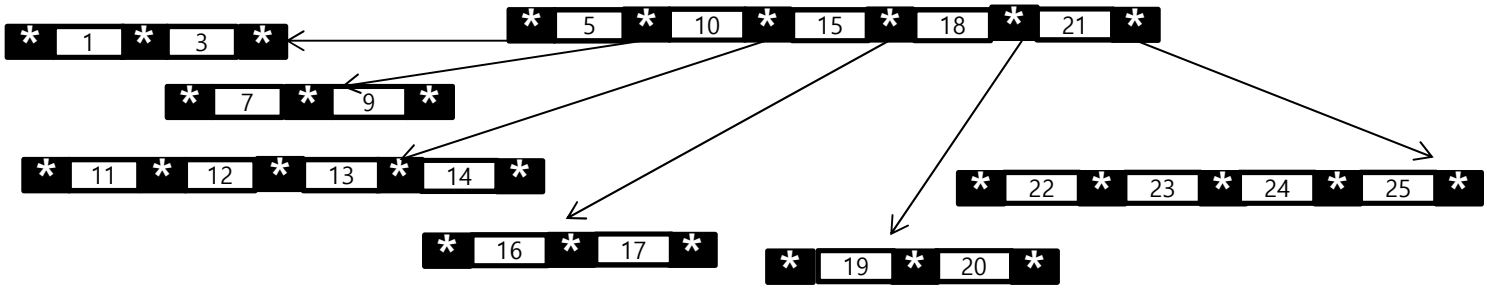
12단계) 15를 삽입하게 된다면 15란 노드를 아까와 마찬가지로 루트 노드로 보내고 16 노드에 대해서 형제 노드로 보냄으로서 형평성을 제공해야겠다.



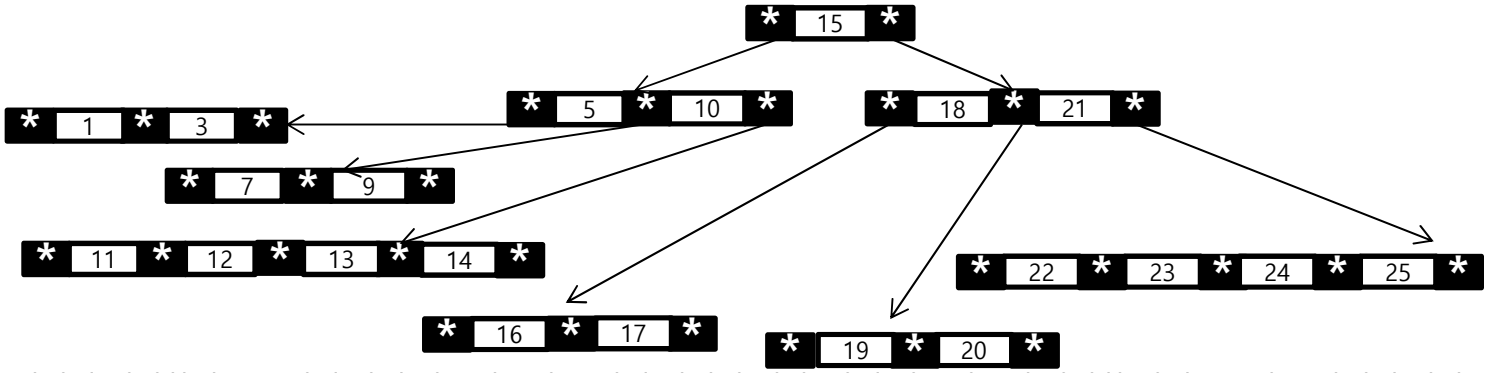
13단계) 17를 삽입하게 된다면 공간이 모든 노드에 대해서 공간이 없기 때문에 루트 노드를 하나 더 만들어서 중간에 18를 넣어줌(중간 인덱스)으로서 정리를 시켜준다.



14단계) 이제 모든 이목구비를 집중해야 할 때이다. 최종적으로 1를 넣게 된다면 어떻게 될까?

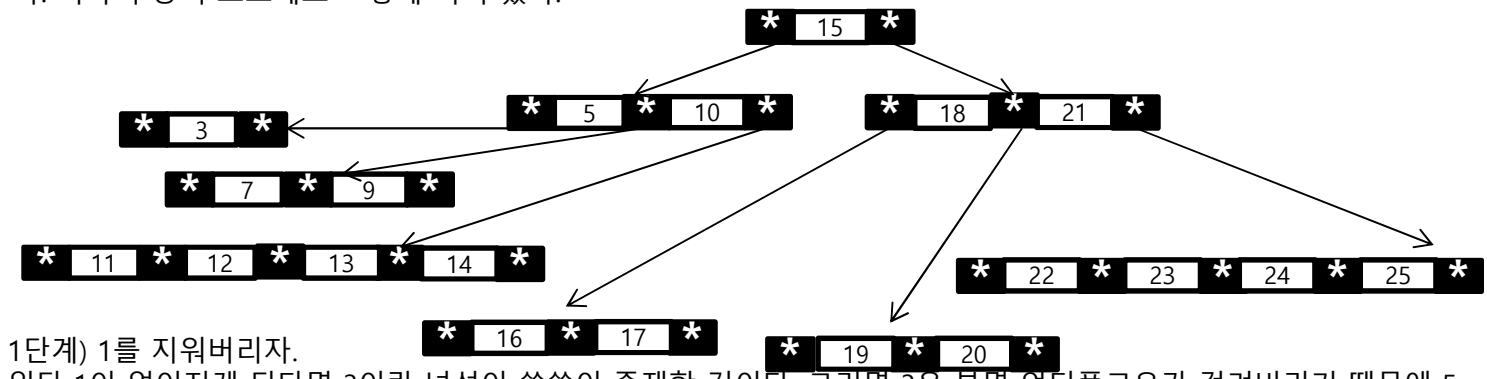


그러면 이제 루트 노드에서 오버플로우가 발생해버린다. 그래서 이에 대한 조정을 하면 되는데 생각보다 쉽다. 루트 노드에 있는 중앙 인덱스인 15에 대해서 위로 올려버리면 되겠다.

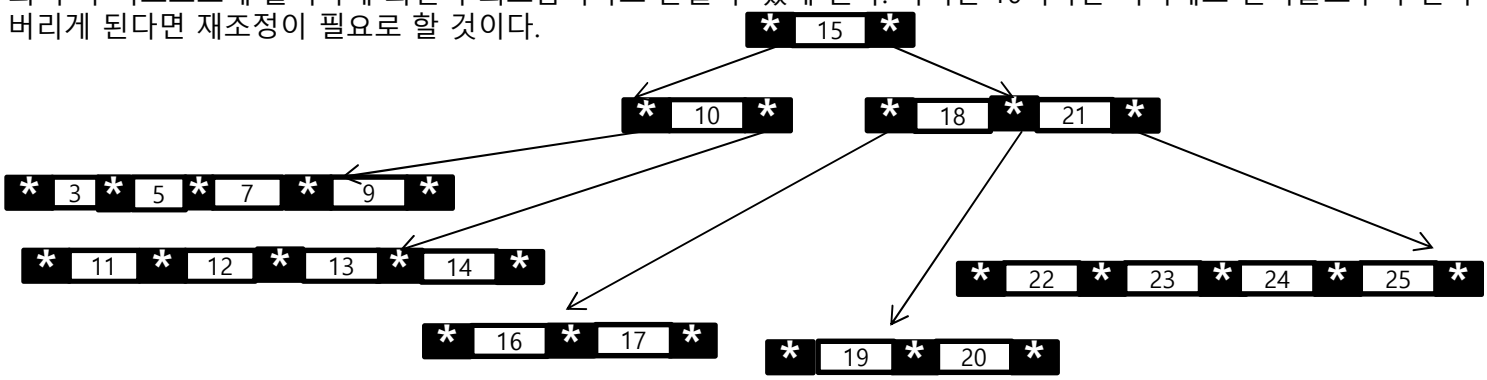


이렇게 정리하면 B-트리의 삽입 알고리즘이 손쉽게 완성이 된다. 삽입 알고리즘의 간략한 방법으로서는 아래에 적어서 후술하겠다.
-> 우선 삽입을 할 여유 공간이 있는지 찾아본다.
-> 오버플로우가 발생시에는 (1) 형제 노드를 살펴봐서 자리에 여유가 있으면 보낸다. 왼쪽부터면 왼쪽 다음에. 오른쪽에 있다면 오른쪽 이전에.
(2) 형제 노드의 여유가 없다면 그냥 가운데 인덱스를 골라서 할당시켜버린다.
-> 여유가 있는 경우에는 맨 끝 값(좌측 서브트리는 오른쪽 값, 우측 서브트리는 왼쪽 값)에 대해서 루트 노드로 보내버리고 여유가 없으면 쿨하게 나누어 버리는 것으로 끝낸다.

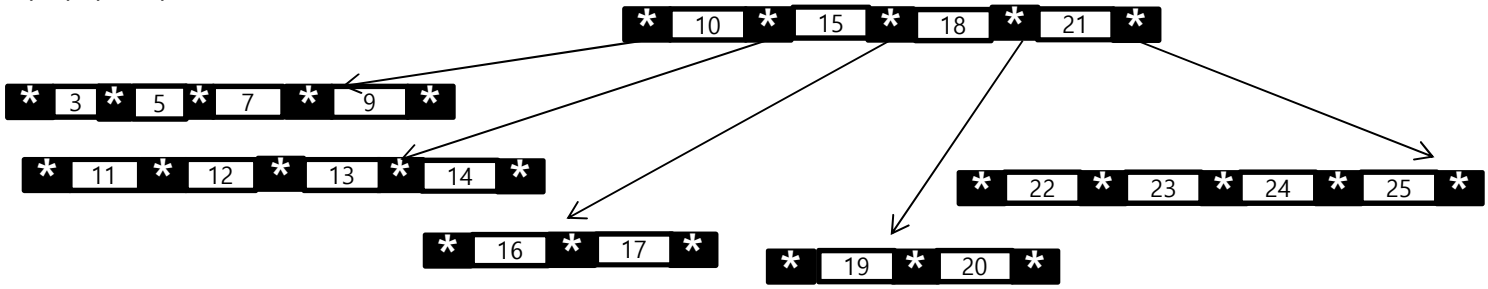
3-3) B-트리 삭제 작동 원리
삽입은 이 정도만 하면 웬만하면 쉽게 이해한다. 그렇지만 좀 애매한 개념이 바로 삭제 작동의 원리이다. 이에 대해서도 연습 문제에 있는 개념을 통해서 삭제하는 과정에 대해 보여주도록 하겠다. 그 전에 언더 플로우에 대해서 알고 넘어가야 겠다.
언더 플로우 : 적은 노드의 용량을 통해서 가리키는 포인터의 용량을 낮추기 위해서 새로이 쓰인 개념. 연습 문제의 사례대로 최소 노드의 수를 2로 잡고 시작을 해야 겠다. 루트 노드에 대해서는 크게 관여를 하지 않고 넘어가보도록 하자. 어자피 강의 노트에도 그렇게 적혀 있다.



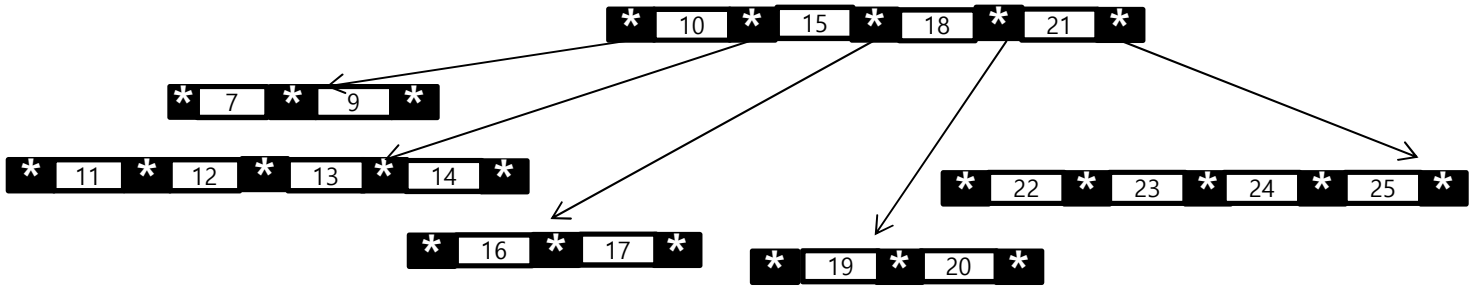
1단계) 1를 지워버리자.
일단 1이 없어지게 된다면 3이란 녀석이 쓸쓸이 존재할 것이다. 그러면 3은 분명 언더플로우가 걸려버리기 때문에 5의 우측 서브노드에 들어가게 되면서 외로움이라도 달랠 수 있게 된다. 하지만 10이라는 녀석에도 언더플로우가 걸려버리게 된다면 재조정이 필요로 할 것이다.



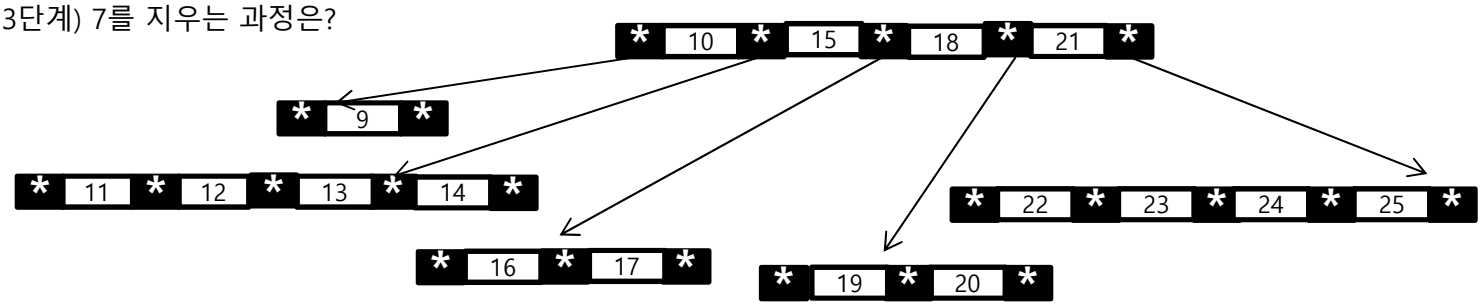
이를 방지하려면 그냥 15라는 녀석을 10 옆에 다시 두는 방법 이외에 없다. 그래서 이렇게 root를 병합함으로서 종결이 나게 된다.



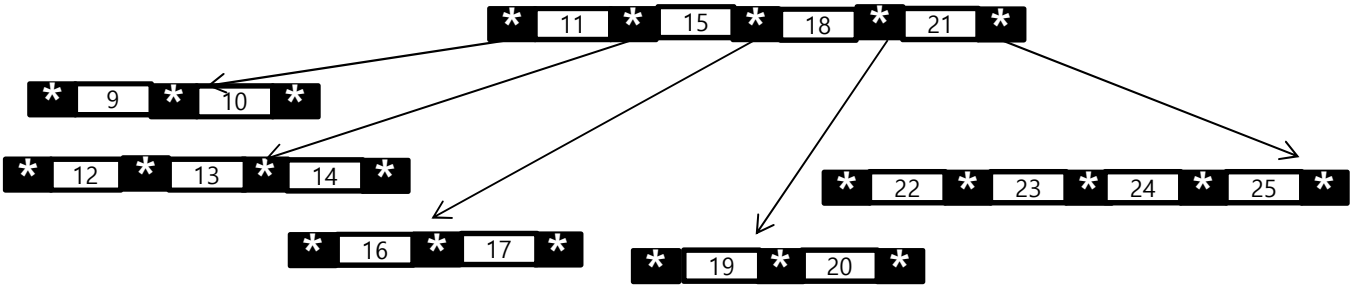
2단계) 3, 5를 지우는 과정은 언더플로우에 지장이 없으니 넘어가도록 하겠다.



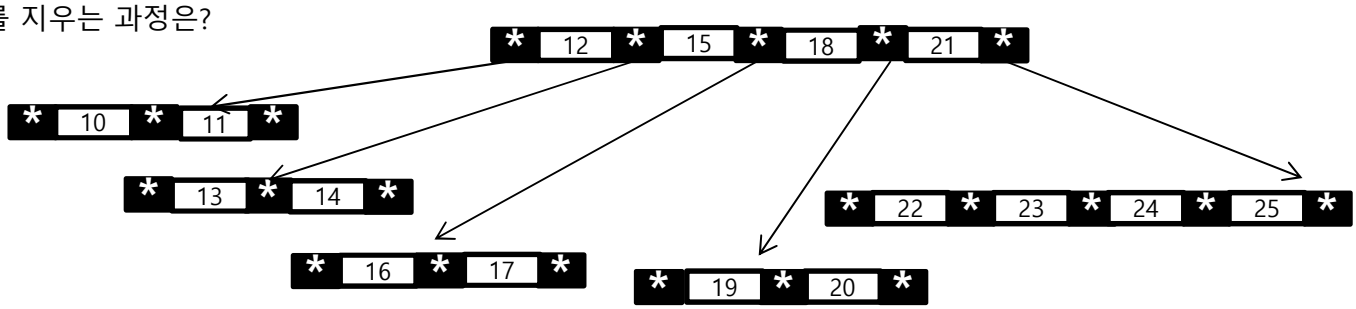
3단계) 7를 지우는 과정은?



일단 7를 없애게 된다면 9는 분명 언더플로우가 걸릴 수 밖에 없다. 9 옆에 10을 넣기 위해서 이웃 형제들에 대해서 공간을 조사를 하고 난 후에 되는 값이 있다면 아래처럼 새롭게 정리를 하면 언더플로우 따윈 개한테 줘버리면 되겠다.

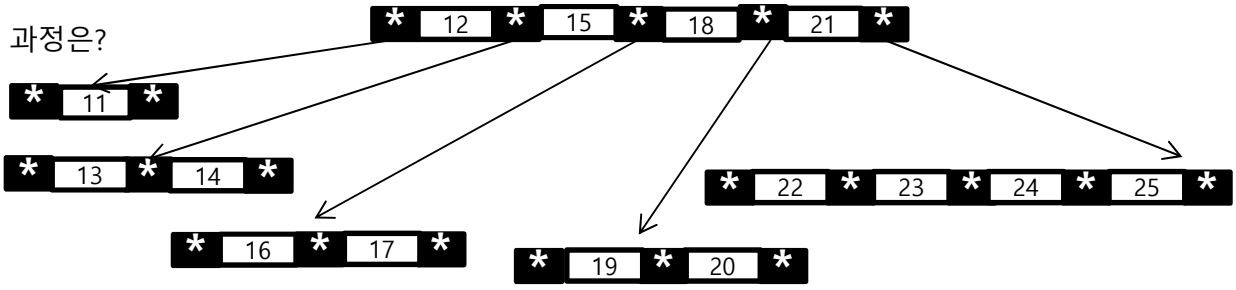


4단계) 9를 지우는 과정은?

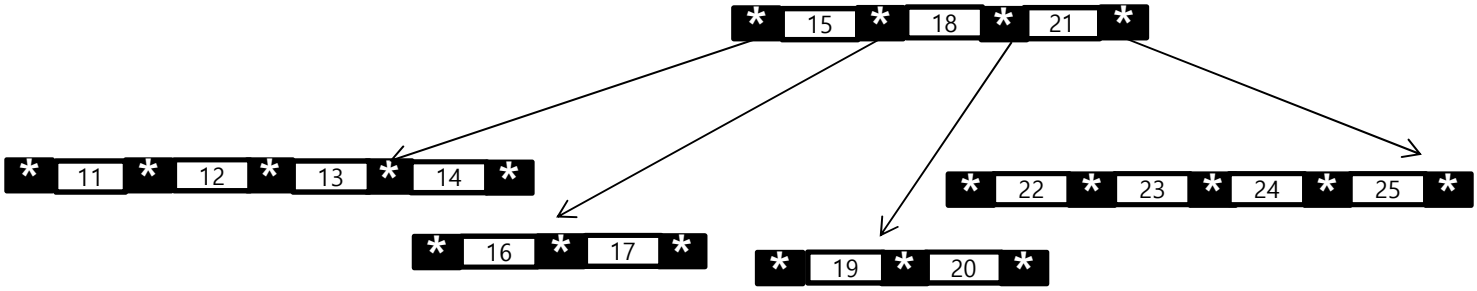


아까랑 똑같은 원리로 가까운 주변 이웃 형제 노드들 중에서 여유가 되는 값에 대해서 넣어줌으로서 삭제를 진행하면 되겠다.

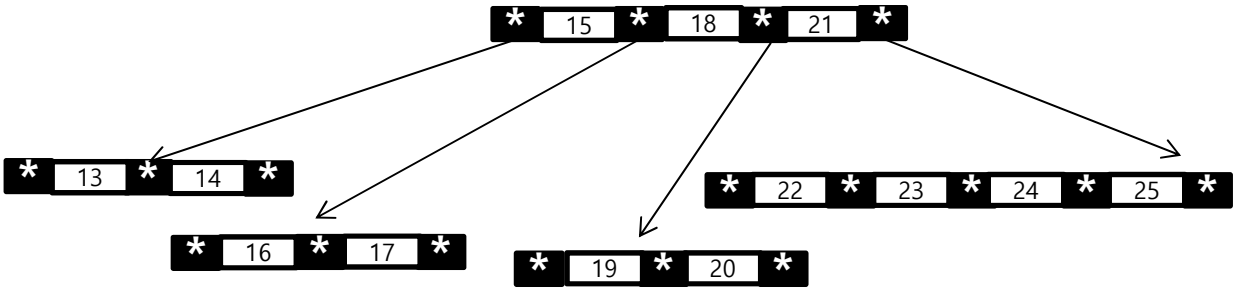
5단계) 10을 지우는 과정은?



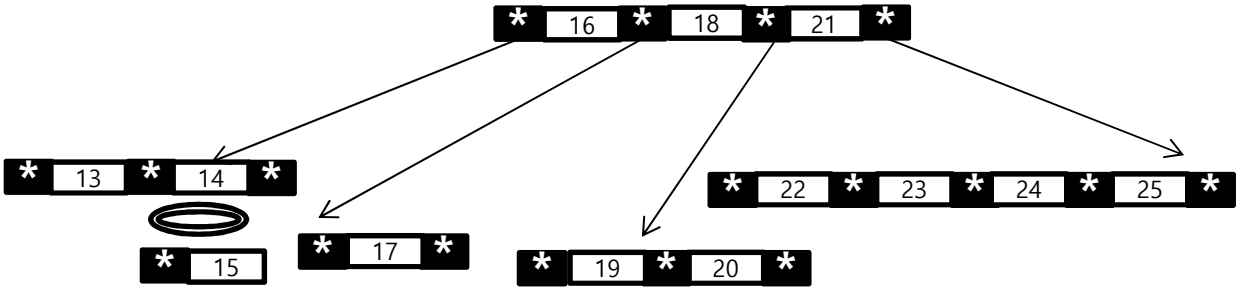
10이 없어지게 된다면 11은 갈 곳이 없게 된다. 그렇지만 13을 다시 옆에 보내게 된다면 더더욱 안될 것이다. 이에 대해서 언더플로우도 생각을 해야 되기 때문이다. 그래서 11이랑 12를 이웃 형제 노드에 다시 삽입을 함으로서 종결을 짓는 거 이외에 없겠다.



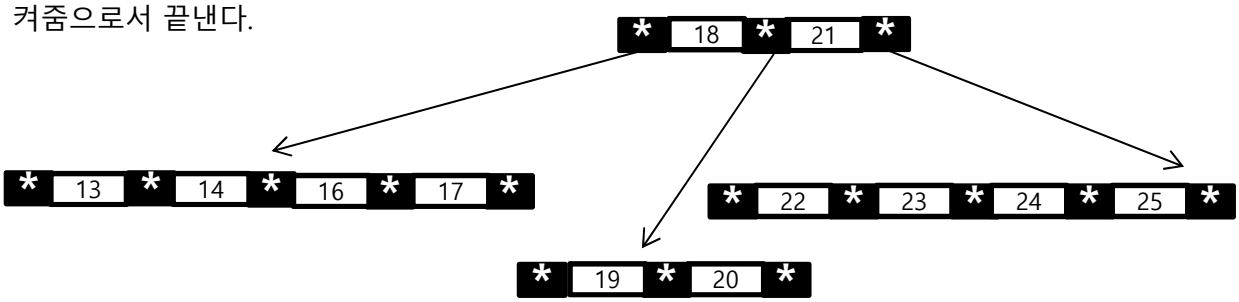
그럼 이처럼 삭제되는 노드에 대해서 정확하게 끝낼 수 있게 된다.
6단계) 11, 12를 지우게 되면 아래처럼 되겠다.



이제 뜬금포 15를 지워보는 이야기를 잠깐 언급해보겠다.
7단계) 15를 지우게 된다면...
우선 16과 15의 위치에 대해서 잠깐 바꿔주도록 하겠다. 그래야지 15를 지우는데 있어서 문제가 일어나지 않는다.



그럼 이제 17에 대해서 언더 플로우가 일어나게 된다. 이에 대해서 13과 14, 그리고 16 이렇게 3개의 개념들을 병합시켜줌으로서 끝낸다.



이로서 B-트리 삭제 알고리즘에 대해서는 아래와 같은 방법으로 요약을 할 수 있겠다.
-> 자식 노드를 삭제하는데 있어서 언더 플로우에 대한 걱정이 필요 없다면 그냥 지워 나가면 되겠다.
-> 하지만 언더 플로우를 의심하게 된다면... 자기 이웃 형제 노드에 대해서 도움을 요청해야겠다.(대신에 오버플로우가 걸리지 않는 범위 이내에서...)
-> 만일 자식이 아니라 루트 노드나 자식의 부모 노드를 삭제한다면 값 사이에 들어갈 만한 숫자끼리 교환을 먼저 해주고 삭제를 해준다.
-> 그리고 언더 플로우가 의심된다면 어쩔 수 없이 병합을 해줘야 겠다.

삭제 알고리즘은 연습 문제를 풀어봄으로서 이해할 수 있을 것이다.

3-4) B-트리 알고리즘의 실체는?
B-트리 알고리즘은 사실상 자바로는 불가능하다고 볼 수 있겠다. 그렇지만 데이터베이스 프로그래밍이나 체계화가 되어 있는 알고리즘을 통해서 구현을 할 수도 있는데 우리는 일단 원리를 이해하는 측면에서 공부한 것으로 만족을 하고 넘어가면 되겠다.